Tables and Arrays in the Analytical Engine

Rainer Glaschick, Paderborn
email & chat: rainer@glaschick.de
2018-12-03

DRAFT

In my paper three years ago on the *Sketch on the Anlytical Engine*, I noted some cases, where assumptions were considered facts because we take something as granted from our today's knowledge.

In the above paper, I observed in particular:

- To overwrite variables, they must contain zero before, unlike any other computer built later.
- Loops were not available in Plan 25, although already considered
- The final example of Bernoulli's numbers lacked some kind of dynamically addressed variables

As concerns the last point, my point of view has changed recently.

Introduction

The *Rosetta* site collects small programming examples in as many different computer programming languages as possible. On example is named the *Babbage Problem* (http://rosettacode.org/wiki/Babbage_problem):

> *What is the smallest integer whose square ends in the digits 269,696?*

In the task description, it is requested that

> *... to do so ... in code that Babbage himself would have been able to read and understand.*

So I began to think about how the programme would read for the AE, provided that simple loops were available.

Most of the examples use the simplest method to calculate the squares of all natural numbers and just compare the tail, sometimes even as strings. This works well for today's increcible fast computers, where 20,000 multiplications and remainder calculations are done in tiny fractions of seconds. Assuming that the AE would require several, say 10 seconds, for multiplications of decimal 10-digit numbers, the machine would need more than 55 hours to solve the problem.

As I wrote in the discussion of the above site, Babbage would never devote days of machine time for this. So my first attempt was to enumerate square numbers by additions only, which is possible by the first (or second) binominal formula, and might reduce the time by the factor 5. Taking into account that the result must have either a 4 or 6 as last digit, the time could be reduced by another factor 5, and thus be more realistic. (I have no idea how quick the division by a power of 10 is to test the tail of a square, but I remember Tim Robinson having mentioned that multiplications and divisions were optimised for cases expected often, and mulitiplications of small amounts should have been fairly quick.)

Still, Babbage would not accept this solution; as the *Rosetta* site reports, he did find one solution, and this surely by paper and pencil. Searching for such a solution[1] it became clear that, once the proper preparation is done — which is necesary for any

programme anyhow — the calcuation itself can be done in less than hour's time with paper and pencil.

This solution, which uses sets of numbers, for which we nowadays use arrays, seems to be outside the AE at first glance, tapping into the above mentioned trap. If we do not implicitly demand that a single programme without input and output should do so, the AE could do the calculations for every desired endings, and even stop if there is no solution. It also provides slightly new views to the task of calculating Bernoulli's numbers.

The algorithm

As mentioned above, by taking the squares of the one-digit natural numbers and observing the last digits, the well-known result is obtained that only a certain set of trailing digits is possible for a square number.

Using Wolframs's notation of `_81` for a number with the last digits 81 (i.e. modulo 10^i if *i* digits follow the underscore), we have:

```
0   1   2   3   4   5   6   7   8   9
0   1   4   9  _6  _5  _6  _9  _4  _1
```

Thus, the solution must end in 4 or 6, if the argument ends in 6. So, the first set contains the numbers 4 and 6, written as

```
{4 6}
```

To find all possible two-digit endings, the squares of all two-digit numbers that end in 4 or 6 are calculated, and the two end digits checked, which results in the set

```
{14 36 64 86}
```

To enumerate the squares of 4, 14, 24 etc, no multiplication is necessary (except the first); and as the numbers are in decimal, no division is required in the paper and pencis method to check the digits.

The same is repeated for 3-digit numbers, giving

```
{236 264 736 764}
```

Two more rounds result in:

```
{0264 2236 2764 4736 5264 7236 7764 9736}
{00264 24736 25264 49736 50264 74736 75264 99736}
{25264 99736 150264 ... (14 more) }
```

The smallest such number is thus 25264. The next larger, 99736, is cited as result from Babbage, who either did not sort the set, stopped at the first solution or had an error in the final caluclations.

The proof that there are infinite many solutions is left to the reader.

Using the AE

Finding the square

Babbage very early planned to read already calculated values from tables on punched cards. (He even insisted that both, the argument and the value, are punched, and that the input mechanism ensures that the argument is compared to protect against wrong cards. We will ignore this sensible measure in the example.)

Using this feature, we can store the data on punched cards in each round and read the numbers in the next round, instead of using arrays.

In pseudo-code (using multiplication for simplicity):

```
read e              -- endings
punch e             -- starts next card batch
read i              -- increment, e.g 100
i * 10 -> m         -- new modulus
punch m             -- this is the next i
e % m -> e          -- part of endings relevant now
while read x from card
    while x < m
        x * x -> y
        y % m -> z
        if z = e
            punch x
        x + i -> x
```

So the first four cards are:

```
269696
    10
     4
     6
```

This produces the second batch (last four numbers sorted for readability):

```
269696
   100
    14
    36
    64
    86
```

Searching the solution in the fifth batch can be done manually.

Note that the first batch can be produced by the two cards:

```
269696
     1
     1
```

The above programme does not reflect that variables need to be zero to be overwritten, and that conditions would just reflect the result of the last operation (i.e. `=0`, `<0`, etc, or overflow). To indicate a variable cleared while obtained, a tilde (`~`) is appended; and using my structured looping (explained in the appendix), assuming that all variables are zero when started, the programme will become:

```
read e                  -- endings
punch e                 -- starts next card batch
read i                  -- increment, e.g 100
i * 10 -> m             -- new modulus
punch m                 -- this is the next i
e~ % m -> e             -- part of endings relevant now
(                       -- endless loop terminated by operator
    x~                  -- clear x
    read x
    m - x -> ~          -- set condition flags for x < m
```

```
        ( > 0               -- skip block unless condition holds
            x * x -> y
            y~ % m -> z
            z~ - e -> ~     -- set condition flag
            ( = 0
                punch x
            )
            x~ + i -> x
            m - x -> ~      -- condition flags
        ) > 0               -- repeat if still x < m
    )^
```

If the tilde is used as destination, it means to discard the result; this might often be a variable not present and depends much on the specific hardware used.

The notation `x~` means clear `x`, e.g. by

```
x~ - x -> ~
```

Also, the hardware may also allow `x~ + ~ -> ~` Or the variable `0` (zero) is does not have any memory and delivers 0 and discards anything, so it may be:

```
x~ + 0 -> 0
```

Of course, the condition as a comparison of two variables could be written just behind the parenthesis, the rule to generate the corresponding comparison is simple:

```
read e                  -- endings
punch e                 -- starts next card batch
read i                  -- increment, e.g 100
i * 10 -> m             -- new modulus
punch m                 -- this is the next i
e~ % m -> e             -- part of endings relevant now
(                       -- endless loop terminated by operator
    x~                  -- clear x
    read x
    ( x < m             -- while x < m:
        x * x -> y      -- calculate square
        y~ % m -> z     -- get ending
        ( z~ = e
            punch x
        )
        x~ + i -> x
    )^
)^
```

Calculating Bernoulli's numbers

The same principle could be used with Bernoulli's numbers. The problem was that the calculation of the n-th Bernoulli number required the values of all previous ones.

So we just punch the next Bernoulli number to a card, add it to the pack, and restart the programme.

As before, each card deck contains some values to tailor the next run, in particular the current index count. The first batch could be created by the first lines of the programme given by Lovelace, and then the batch of programme cards would just be started over.

To provide a new programme would require to find a notation for card read and punch, and thus would use a textual representation.

I remember someone to propose to use the table given by A.L. as a template, which comes close to my new view.

Remarks

Note that this technique was intensively used with the Pilot ACE, as its memory was not large enough to hold larger matrices. And as Pilot ACE used punched cards, it could read numbers at least one magnitude quicker that paper-tape based machines, and — as the fastest machine in the word for years — solve problems that were not feasible on other machines.

Also note the Zuse's Z3 did neither have arrays nor have a loop instruction, so he would have been in a similar situation. Zuse wrote that the programme tape could be glued to a loop if necessary. Only the card or tape reader and punch for data were missing to solve this problem.

Appendix

Details on the method used

May be provided

A Bourne shell script solution

May be provided

Structured conditions and loops

A structured notation for loops and conditions is presented, that is equally appropriate for sequential and random access programme storage.

In sequential programme storage, it is assumed that forward and backward jumps do not need counts; instead, programme cards contain labels, and the cards are skipped until the label is found. For the following notation, the label is just the nesting level of the parenthesis. Indentation also indicates the nesting level.

Any conditional or loop block has a pair of parenthesis, that surrounds a statement block:

```
(                   -- open a block
    ....            -- statement block
)                   -- close a block
```

Without a condition following the parenthesis on the same line, the parenthesis is just a label, i.e. the block entered and and left without any control change.

If there is a condition — whatever is possible in the machine or language --, the actions are inverted as follows:

- For an opening parenthesis, if the condition is true, nothing is done, i.e. the block is entered; otherwise a forward skip starts until the corresponding closing parenthesis is found. The forward skip may or may not
- For a closing parenthesis, if the condition is true, a backward skip starts until the corresponding open parenthesis is found; otherwise nothing is done, i.e. the next instructions after the parenthesis executed. The backward skip may or may not check again the condition; as it is inverted anyhow, the block is always entered

from a backward skip.

A forward or backward skip may or may not check again the condition; as the actions are inverted, a forward skip will not repeat, and a backward skip will enter the block.

Thus, if a instructions are to be executed if the last calculation was zero, the notation is:

```
( =0
    ...
)
```

An old-fashioned do-loop to repeat while the last calculation is greater zero reads:

```
(
    ....
) >0
```

A modern while-loop with the same condition becomes:

```
( >0
    ...
) >0
```

For better readability, the caret (hat, ^) may be used after a closing parenthesis, which means *same condition as on corresponding opening parenthesis*):

```
( >0
    ...
)^
```

The caret is also needed for an endless loop:

```
(
    ...
)^
```

With a random access programme, there is no problem to jump back on the top, just to find out, that the condition no longer holds, and skip the block again. With sequential store, the skip back can be avoided, just be repeting the condition.

If an alternative (*else*) is needed, this would be coded as:

```
( =0
    ...
)
( ~0
    ...
)
```

Thus, the condition has to be inverted. For better readability, both parenthesis can be set on the same line:

```
( =0
    ...         if zero
)(
    ...         if not zero
)
```

The two shortcuts are easy to handle, e.g. by a clerk punching cards, as just the

counterpart has the be found.

At the first glance, it might be strange that to determine if a block is a loop or an if, one has to look at the end.

Of course, the operation that sets the condition has to be repated for a while-loop, as in

```
m - x -> ~
( > 0
    ...
    m - x -> ~
) > 0
```

In order to repeat or leave a loop, the caret (or less) and greater signs can be used:

```
(       -- endless looo
    ...
    ^       -- repeat unconditionally
    ^ =0    -- repeat if condition holds
    < =0    -- same as above
    ^^      -- repeat two levels
    >       -- exit
    > >0    -- exit if last calculation was positive
    >>      -- exit two levels
```

This does not require any special means, even not on sequential programme storage; it is just a (conditional) skip back to a label.

References

Rosetta:
Rosetta site: *Babbage Problem* https://rosettacode.org/wiki/Babbage_problem

---

[1]with a little help from my wife who is a very capable in number crunching without computers, knowing many tricks and quickly seeing patterns